Overlapping Multi-Processing and Graphics Hardware Acceleration: Performance Evaluation

Xavier Cavin* Laurent Alonso Jean-Claude Paul

LORIA† - INRIA Lorraine

Abstract

Recently, multi-processing has been shown to deliver good performance in rendering. However, in some applications, processors spend too much time executing tasks that could be more efficiently done through intensive use of new graphics hardware. We present in this paper a novel solution combining multi-processing and advanced graphics hardware, where graphics pipelines are used both for classical visualization tasks *and* to advantageously perform geometric calculations while remaining computations are handled by multi-processors. The experiment is based on an implementation of a new parallel wavelet radiosity algorithm. The application is executed on the SGI Origin2000 connected to the SGI InfiniteReality2 rendering pipeline. A performance evaluation is presented. Keeping in mind that the approach can benefit all available workstations and super-computers, from small scale (2 processors and 1 graphics pipeline) to large scale (p processors and n graphics pipelines), we highlight some important bottlenecks that impede performance. However, our results show that this approach could be a promising avenue for scientific and engineering simulation and visualization applications that need intensive geometric calculations.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing ; I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors ; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Radiosity

Keywords: Parallelism, Graphics Hardware, Hierarchical and Multiresolution Algorithm, Wavelet, Realistic Rendering, Radiosity.

1 Introduction

1.1 Motivation

Recent commercial ccNUMA super-computers, like the SGI Origin2000 [15], have been shown to perform and scale for a wide range of scientific and engineering applications [13]. It is important to evaluate their strengths and weaknesses for graphics rendering applications. Recent papers have proven that parallelism makes it possible to deal with extremely large geometrical models in a reasonable time within some rendering applications, such as realistic rendering [19, 20, 4, 18] or volume rendering [1]. On the other hand, newly available graphics hardware now also provide high performance for rendering tasks. In some ways, since multi-processors speed up rendering computational times, they create the need for software that replicates the functionalities of graphics hardware. In a previous paper [4], we presented an efficient parallelization of a wavelet radiosity algorithm and we pointed out this limitation of parallelism for radiosity-based rendering.

*615 rue du Jardin Botanique, BP 101, F-54602 Villers-lès-Nancy Cedex, France. Xavier.Cavin@loria.fr

†LORIA is UMR 7503 LORIA, a joint research laboratory between CNRS, Institut National Polytechnique de Lorraine, INRIA, Université Henri Poincaré and Université Nancy 2.

1.2 Related work

Wavelet radiosity is a very efficient algorithm introduced by [12] and [22]. As with all hierarchical N-Body methods, this algorithm has highly *irregular* and *unpredictable* data access patterns that make its parallelization challenging [25]. A few papers [11, 17, 5] have proposed a parallel implementation of hierarchical radiosity for large models. Wavelet radiosity is also a hierarchical approach, but it provides a more formal mathematical framework for radiosity. Thus, while dealing with high level order basis functions, it is able to compute highly precise radiosity solutions. It is important to point out that, in order to get this high precision, it must perform much more interaction calculations between the finite elements, compared to the classical hierarchical approach.

Despite this difficulty, we implemented a parallel radiosity algorithm, based on linear wavelet bases, on the SGI Origin2000 and obtained significant speed-ups. For instance, we ran our application on the well-known Soda Hall building test model in order to render a realistic walk-through. A radiosity solution with 1 700 000 patches (compared to the 143 097 initial surfaces) was computed in 15 hours with 16 processors, while it would take about 7 days on a single processor.

As previously pointed out, in our multi-processors environment, we had to use software-based computations that replicate the functionalities of available graphics hardware. Thus, we were curious to evaluate how a sequential wavelet radiosity algorithm, benefiting from advanced capabilities of recent graphics hardware - like *pbuffers* - could perform.

In the radiosity literature, almost all implemented algorithms rely on software. Nevertheless, one of the first implementations of the radiosity method used a depth buffer to perform visibility and form factors calculations between surfaces in the iterative radiosity solving process, either in a pre-processing step [7] or during computations [6], and the authors suggested that dedicated graphics hardware could be devoted to this task. Recently, two papers have also proposed using graphics hardware in realistic radiosity [14, 16], but the use of hardware is devoted to the ultimate rendering pass, not to the kernel evaluation of the radiative process.

In [2], we proposed acceleration techniques, taking advantage of existing graphics hardware, to speed-up the visibility queries. These techniques are already advantageously integrated inside our sequential wavelet radiosity algorithm. For the purpose of the present paper, we have tested this sequential program on a SGI Onyx machine connected to the SGI RealityEngine2 and rendered the Soda Hall model in about 34 hours. These results are similar to those that would be obtained on the SGI Origin2000 with 8 processors. In other words, comparable speed-ups have been obtained with a sequential implementation taking advantage of graphics hardware acceleration, than with a parallel implementation where a lot of time is spent performing intensive geometric calculations.

The new algorithm presented in this paper has been designed to take advantage of both multi-processing *and* graphics hardware acceleration. In the radiosity rendering process, graphics hardware

is commonly used, similarly to other visualization applications, at the final visualization step. We obviously do this also. The key difference is that we also use graphics hardware for accelerating some costly computational tasks - the visibility queries - during the radiosity equation solving process.

We must note that this idea has been previously introduced in [3]. In this paper, the authors have proposed a method that enhances radiosity algorithm performance by using the capabilities of a multi-processors graphics workstation: hemicube items are produced by a master processor using the graphics hardware, as in [7], while the remaining computations are performed in parallel on the remaining processors. Our algorithm differs from this solution in three ways:

- our wavelet radiosity algorithm more efficiently performs highly precise solutions than the classical algorithm proposed;
- using a clever visibility algorithm, our implementation does not suffer from all the drawbacks of the standard Z-buffer hardware;
- we do not use a master-slave scheme, thus allowing our parallel implementation to scale and deal with various combinations of processors and graphics pipelines.

However, the approach we propose is very similar in spirit.

1.3 Aims and organization of the paper

In summary, the advantages of our new algorithm are:

- our novel wavelet radiosity algorithm is of complexity $O(n)$ [21], it can deal with high order basis functions and render extremely large models;
- our parallel implementation allows both acceptable load balancing and excellent data locality, that permits the scaling of the wavelet algorithm to a large number (64) of processors;
- the graphics hardware computes the visibility calculations required by the wavelet radiosity solver between 2 and 10 times faster than a software package and moreover, frees the multiprocessors for other computational tasks.

The novel algorithm was implemented on the SGI Origin2000, which is a ccNUMA super-computer, with MIPS R10000 processors and an aggressive, scalable distributed shared memory (DSM) architecture. This machine was configured for the experiment with a single SGI InfiniteReality2.

To validate our approach, the new algorithm has been tested on the classroom reference test scene, made available by Peter Shirley for the 5^{th} Eurographics Workshop on Rendering. Using this model, we measured the performance of our new scheme. We also highlight the most important bottlenecks that come in the way of performance, keeping in mind that our new scheme could benefit all available graphics workstations, from small scale (2 processors and 1 graphics pipeline) to large scale (p processors and n graphics pipelines). Finally, we tested our algorithm on the Soda Hall model.

In Section 2, we briefly recall general concepts on wavelet radiosity and show how its structure offers some insight to partitioning the algorithm across multiprocessors and graphics hardware. Section 3 shows how we advantageously use parallelism to perform this algorithm, while Section 4 presents how graphics hardware acceleration could be successfully applied in a sequential implementation. Then, in Section 5, we show how overlapping multiprocessing and graphics hardware acceleration can enhance parallel realistic rendering. Performance evaluation and results are presented and discussed in Section 6 and a general discussion is exposed in Section 7. Finally, Section 8 concludes and presents future work.

2 Wavelet Radiosity Algorithm Structure

Let us first recall, for the sake of clarity, the radiosity equation, and more specifically the terms involved in it and their implication to the parallelization process.

The radiosity - power per unit area $[W/r^2]$ - on a given surface is defined as the light energy leaving the surface per unit area. Let $\mathcal{M}$ denote the collection of all surfaces in an environment, and χ be a space of real-valued functions defined on $\mathcal{M} \times \Lambda$; that is, over all surface points and all wavelengths (Λ). Our goal is to determine the surface radiosity function $f \in \chi$ that satisfies:

$$f(\lambda, x) = g(\lambda, x) + k(\lambda, x) \int_{\mathcal{M}} G(x', x) f(\lambda, x')\, d\, x', \qquad (1)$$

where λ is the wavelength at which functions are computed.

2.1 The wavelet radiosity solver

The $\mathcal{M}$, g and k terms

These three terms represent the input data of a radiosity simulation problem:

- $\mathcal{M}$ consists of the set of input surfaces. They can be either simple triangles or polygons, or more complex shapes;
- $g(\lambda, x)$ specifies the origin and directional distribution of emitted light. It either represents point light sources or self-emitting surfaces from $\mathcal{M}$;
- $k(\lambda, x)$ is the local reflectance function of the surface. Within the radiosity assumption, we suppose that the surfaces are ideally diffuse (*i.e.* lambertian surfaces).

The $\mathcal{M}$, and to a lesser extent, g and k terms are responsible for the *irregular* characteristic of the physical domain being simulated. The initial distribution of surfaces and emitted energy make it difficult to anticipate how and where the computational power will be focused: distributing it in parallel will be even more challenging.

The f term

The $f(\lambda, x)$ term is the radiosity function we aim to compute. In order to make it calculable, f must be projected in a finite dimension subspace. Let $\chi_n \in \chi$ be a subspace of χ of dimension n and $\{\phi_i\}_{i \in \{1,\ldots,n\}}$ one of its bases, we now have to determine the $\{\alpha_i\}_{i \in \{1,\ldots,n\}}$ coefficients, so that:

$$f \approx f_n = \sum_{i=1}^{n} \alpha_i \phi_i.$$

The choice of the basis functions is very important, because it greatly influences both the the precision of the approximated function and the complexity of the computations. Historically, the first radiosity algorithms used piecewise constant functions. For efficiency reasons, more recent algorithms have introduced hierarchical and wavelet [12, 23] functions.

Using hierarchical wavelet bases, however, gives to the algorithm a *dynamic* and *unpredictable* characteristic. Indeed, input surfaces of $\mathcal{M}$ are subdivided as the computation proceeds, making it impossible to foresee both the time needed to compute an energy transfer and the total amount of memory required (*i.e.* memory must be dynamically allocated). Obviously, this becomes even more problematic in parallel.

The $\int_{\mathcal{M}}$ integral

The integral equation codes the interactions between all surface points of the input scene. The resolution process solving this equation is closely related to the different terms of Equation 1 and uses them to make the needed computations.

The resolution can be tackled using two main approaches, called *gathering* and *shooting*. The two approaches are based on Gauss Seidel and Southwell iterative methods, which are very well described in [8] or [24]. In the first one, each step in an iteration updates the coefficient of one basis function, by accumulating the energy of all others basis functions. A total iteration consists of the set of necessary steps to update all basis functions. The convergence rate can be accelerated by the shooting algorithm, allowing it to draw intermediate results quickly. Each step updates all basis functions coefficients by propagating the basis function with the most energy. The convergence is faster, but the computation errors are propagated along with the resolution.

The two approaches exhibit common problems concerning their parallelization. First, care must be taken when accessing and updating data structures in parallel. Then, the work of propagating energy between basis functions has to be distributed among processes in a well load-balanced way. Finally, data accesses must be done efficiently in order to get good data locality.

2.2 The G term

$G(x', x)$ is a geometry term defined by:

$$G(x', x) = \frac{\cos\theta_{x'} \cos\theta_x}{\pi\, r^2_{x'x}}\, v(x', x),$$

where r is the length of the vector connecting the two surface points x and x', and θ_x and $\theta_{x'}$ are the angles between the local surface normals and this vector (see Figure 1).

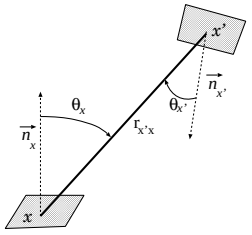


Figure 1: Components of geometry term G.

The right part of the G term is the visibility function $v(x', x)$, that gives it a global scope. Indeed, the visibility function may involve any other surface, not just the two interacting ones. This function is responsible for most of the discontinuities in the final solution f, and so shall be as exact as possible, with respect to the precision of f. Visibility accuracy has a cost and may be responsible for the largest part of the overall computation time, especially in the wavelet radiosity algorithm.

We must note that in the wavelet radiosity algorithm, the criterium for dividing an interacting surface element, the *oracle* function, involves point to point visibility computations between sample points on the emitting (x^e_k) and receiving (x^r_k) surface elements (Figure 2): at *each* level of a given interaction, we have $n_e * n_r$ visibility requests to evaluate. Visibility calculations clearly have to be optimized in order to get efficient algorithms.

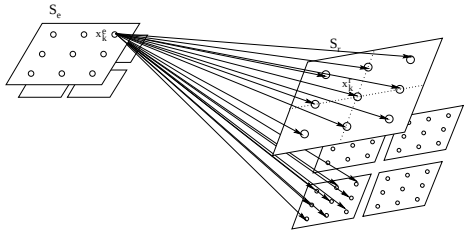


Figure 2: Visibility requests between surface elements.

3 Taking advantage of parallelism

In [4], we presented a parallel radiosity algorithm based on the following:

- the f function is projected onto wavelet bases;
- the $\int_{\mathcal{M}}$ integral is solved using a progressive shooting algorithm that does not store links between surfaces for memory requirements and programming ease considerations.

  This parallel algorithm can be decomposed into two successive parts, direct illumination by light sources followed by redistribution of energy between surfaces, that will be detailed below;
- the v function is implemented with a BSP accelerating structure.

  The most trivial way to handle a point to point visibility request is to draw a ray between the two points and then check all scene surfaces for intersection. A straightforward implementation of this algorithm suffers from an $O(n)$ complexity, and cannot be used in practice. Fortunately, algorithms based on spatial subdivision are mature enough to have become a standard and are now being widely used to accelerate such ray-tracing requests. We have chosen a Binary Space Partition structure (BSP) suggested by [10] as an accelerating structure.

Results of this implementation are given in Section 6.

3.1 Direct illumination

This phase is responsible for direct lighting effects and for initial energy distribution before the inter-reflections computation. In our case, we deal with C/γ representations [9] coming from industrial measures. Point light sources are processed one by one. We first determine the set of surfaces visible from the light position within its spatial distribution of emitted light. We then recursively illuminate each of these surfaces, subdividing the so created surface elements when the quality of the approximation has to be improved.

We have chosen, for parallelizing this phase, to decompose it into a pool of tasks, each consisting of an interaction between a light source and an initial surface. These tasks are distributed to processes in the same order they would be processed by the sequential algorithm. When a receiving surface is already handled by another process, the process has to wait for the energy transfer to be completed. We plan to enhance this strategy in order to avoid unnecessary *idle* time.

3.2 Inter-reflections

Once the direct illumination phase is completed, we build a list of potential emitting surfaces, sorted by decreasing energy. We then iterate over the surfaces of this list, choosing at each step the surface, S_e, with the most energy. We determine the set of surfaces visible

from S_e and for each surface, S_r, in it, we recursively propagate the energy, as was done in the direct illumination phase, except that now, both surfaces can be subdivided into surface elements (Figure 2). Once the energy transfer is computed, the updated receiving surface, S_r, is inserted into the sorted list.

This phase has been parallelized in two ways, one using the inner loop of the sequential algorithm, that is for a given emitting surface, S_e, the loop over the receiving surfaces, S_r, and the other using the outer loop, that is the loop over emitting surfaces, S_e. Both approaches have their drawbacks (synchronization barriers for the first one, restrictive access to the receiving surface and lower convergence rate for the second one), but can be advantageously combined to minimize their drawbacks: time consuming most energetic surfaces can be handled with the inner loop parallelization at the beginning of the resolution, while the final interactions can be quickly computed by the outer loop parallelization until convergence.

4 Taking advantage of graphics hardware

The radiosity algorithms presented in Section 3 contain two main kinds of visibility requests:

- point to environment or surface to environment requests, *i.e.* what surfaces are potentially visible, from a point or a surface? These requests consist of "clipping" the subset of input surfaces visible from the current emitter;
- point to point requests, *i.e.* is a point visible from another point?

Our implementation of the BSP-based visibility algorithm presented in Section 3 can only answer point to point requests. Thus, all scene surfaces are potential receivers for a given emitter and must be checked for visibility. Even despite the spatial accelerating structure, all these visibility queries are still responsible for too huge a part of the computation time.

Another solution to speed up visibility computations is to turn toward projective approaches. This can be achieved by using the new advances in both graphics workstation architectures and graphics libraries, that allow programmers direct and fast access to the results of graphics boards computations, such as the Z-buffer algorithm.

Let us see how visibility requests can be efficiently answered using this approach, and how this impacts on the computation times, and consequently on the previously achieved parallel speed-up. A more detailed presentation of hardware-based visibility algorithms can be found in [2]. Results of this implementation are given in Section 6.

4.1 Hardware visibility

This method is inspired from the hemicube method, originally devoted to computing form factors in classical radiosity.

The original hemicube method starts by associating each input surface with a unique color[1], keeping one for the background. It then builds a hemicube of 5 pixmaps centered over the point of interest. A rendering of the input scene, using the false associated colors, is done on each pixmap, using the Z-buffer algorithm for hidden surface removal. Figure 3 illustrates this step.

It is then trivial to approximately determine which surface is visible from the point of interest in a given direction by querying the pixel corresponding to that direction: its color uniquely defines the visible surface (or the background if no surface is visible).

[1]This limits to $2^{24} - 1 = 16777215$ the number of input surfaces on a classical 24 bits color display!

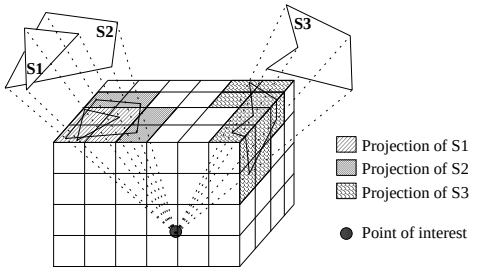


Figure 3: Hemicube of Z-buffers.

We have developed two algorithms based on this technique, depending on whether we are in the direct illumination phase (point-based visibility requests) or the inter-reflections phase (plane-based requests).

Point-based requests

During the direct illumination phase, all visibility requests start from the same point, for a given light source. Since the spatial distribution of emitted light can be spread over all directions, we no longer use a hemicube (Figure 3), but rather build a complete cube centered over the current point light source. We are thus able to quickly clip the surfaces visible from the light source (point to environment visibility request) by building the set of surfaces corresponding to all pixels found on the 6 pixmaps. It is also immediate to determine if a surface point is visible from the light source (point to point visibility request) by finding the pixel intersected by a ray traced between these two points and comparing its color to the surface false color.

Plane-based requests

If the point-based visibility algorithm is well-adapted to the direct illumination phase, it is not the case for the inter-reflections phase. Indeed, initializing a hemicube of pixmaps is not free of computation time and really only has interest if it can be reused for several requests. It does not seem reasonable to use a hemicube for each sample point on the current emitting surface element: the probability of being reused many times would be too low.

Instead, we set up a regular grid on each input surface, with each grid point being a potential point of interest for a hemicube of pixmaps. Figure 4 shows a configuration for an input surface, with some hemicubes being built and the others not.

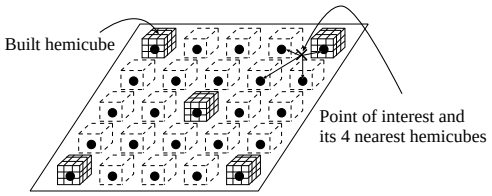


Figure 4: Regular grid of hemicubes.

Clipping the visible surfaces from the emitting surface (surface to environment request) may be achieved in a different manner: we may render the input scene either on the central hemicube, or on the central hemicube and on the 4 corners (the case in Figure 4), or finally on all grid point hemicubes. It clearly depends on the size of the surface: the number of hemicubes must be higher for a huge building floor than for a single pen on a table! The final set

of visible surfaces is simply the union of the visible surfaces of all hemicubes.

We determine the answer to point to point requests by approximating the origin of the request (the emitting surface element point) by the nearest grid point. The hemicube associated with this grid point may either be built from previous computations or not. In the negative case, we initialize it with the grid point as a point of interest. We then use this hemicube for querying the pixel in the original direction and determine visibility.

4.2 Hybrid visibility

Hardware visibility has some inherent drawbacks, due to the discretization implied by the pixmaps of the hemicubes. In particular, some aliasing phenomena may appear and small objects may be missed. Moreover, if the hemicube is composed of N pixels, at most N scene surfaces may be detected as visible. Visibility errors clearly depend on the pixmaps size (anyway limited to a maximum size of about 1000×1000), which must be adapted to the scene size. In practice however, pixmaps sizes between 100 and 500 are usually enough to avoid perceptually visible errors on the final result.

We have defined heuristics to evaluate the confidence rate of a given answer of a hardware visibility request. If we detect that this could potentially be a false answer, we can resort to another visibility algorithm, such as the BSP-based visibility one presented in Section 3. In this way we are able to combine the quickness of the hardware visibility algorithm with the correctness of the BSP-based visibility algorithm.

We briefly explain here how the new hybrid algorithms answer point to point visibility requests.

Point-based requests

We start a point-based hybrid request as a point-based hardware request. The difference is that we not only query the pixel in the given direction, but also its 4 nearest pixels. We then compute, for each corresponding surface (if any), its intersection with the ray joining the two points, and we keep the nearest one. At this point, if we found a surface, we consider it as the one visible. In the other case, we suspect a visibility problem and decide to use a BSP-based visibility request to retrieve the result.

This solution leads to a large number of BSP-based visibility requests in problematic places. To avoid this, we check if a significant part of the previous visibility requests could not be answered using the hemicube: in this case, we enhance the resolution of the hemicube by zooming in on the problem area. We do this by building a new pixmap centered on the problem area, with the same size as the hemicube pixmaps, but twice the resolution. We now use this pixmap as long as visibility requests fall inside it. This process can be recursively repeated on very problematic places. Once a pixmap is no longer used, it is deleted and replaced by the previous one.

Plane-based requests

Plane-based hybrid visibility requests are a little more complicated. Indeed, since we approximate the origin point with the nearest grid point as the center of the hemicude, the previous coherency test and the zoom mechanism have less meaning.

We rather try to exploit the fact that the origin point is located on a surface. For that purpose, we determine the 4 nearest grid points - sometimes just 2 or even 1 if the point lies on the surface border - and use their associated hemicubes, building them if necessary (see Figure 4). We then make a coherency test between pixels, and associated surfaces, sampled on these hemicubes and do a BSP-based visibility request if it fails. A complete description of this test may be found in [2].

5 Combining parallelism and hardware acceleration

We have seen in Section 3 that radiosity computations could be accelerated with multi-processing. Then, in Section 4, we have seen that it could also be accelerated, in a sequential implementation, by using graphics hardware to handle visibility computations. It seems natural to try to combine these ideas in order to take advantage of their respective performance gains.

We consider for the remainder of this Section a multi-processors and multi-pipelines configuration.

Basically, the job that processes execute in parallel can be summarized by Algorithm 1, where e denotes either a point light source or a scene surface.

Algorithm 1 Simplified algorithm of a process

```
1: while e ← getNextEmitter(e) do
2:   while r ← getNextReceiver(e) do
3:     computeInteraction(e, r)
4:   end while
5: end while
```

Visibility computations occur at two different stages of this short algorithm. On one hand, for clipping the subset of potential visible surfaces: when a process gets its next emitter, through the $getNextEmitter()$ function described by Algorithm 2, it may be the first one to request it. In that case, the list of potential visible surfaces associated with that emitter has to be initialized. If the emitter requires a hardware-based visible surface clipping, the process has to wait for an available graphics pipeline to compute it with the clipping algorithms presented in Section 4. Otherwise, it computes the subset of visible surfaces through a software-based algorithm[2].

Algorithm 2 Getting the next emitter

```
Require: the previous handled emitter prev_e
 1: e ← chooseNextEmitter()
 2: if e ≠ prev_e and e → visibleSurfaces = ∅ then
 3:   if e → requiresHardwareClipping then
 4:     e → lock()
 5:     if e → visibleSurfaces = ∅ then
 6:       getAvailableGraphicsPipe()
 7:       l_c ← buildVisibilityCubesForClipping(e)
 8:       releaseGraphicsPipe()
 9:       (e → visibleSurfaces) ← (l_c → getSurfaces())
10:     end if
11:     e → unlock()
12:   else
13:     (e → visibleSurfaces) ← getVisibleSurfaces()
14:   end if
15: end if
```

On the other hand, once the process has its new emitter-receiver interaction and is processing it through the $computeInteraction()$ function, it is likely to have to handle point to point visibility requests to achieve the computation as explained in Section 2. The point to point visibility algorithm is given by Algorithm 3.

The first thing a process has to do before computing a point to point visibility is to ensure that all the necessary visibility cubes (a single cube in the case of a point-based request, 1, 2 or 4

[2] In our case, we simply return the set of all input surfaces.

Algorithm 3 Computing a point to point visibility request

Require: two points pt_e and pt_r
1: $l_c \leftarrow getVisibilityCubes(pt_e)$
2: **if** ! $(l_c \rightarrow upToDate())$ **then**
3: $getAvailableGraphicsPipe()$
4: **for** each $c \in l_c$ **do**
5: **if** ! $(c \rightarrow upToDate())$ **then**
6: $c \rightarrow lock()$
7: **if** ! $(c \rightarrow upToDate())$ **then**
8: $c \rightarrow build()$
9: $c \rightarrow setUpToDate()$
10: **end if**
11: $c \rightarrow unlock()$
12: **end if**
13: **end for**
14: $releaseGraphicsPipe()$
15: **end if**
16: $computeVisibility(pt_e, l_c, pt_r)$

hemicubes for a plane-based request) have been built. If not, it has to wait for an available graphics pipeline in order to build them one by one. It is important to note that the process keeps the graphics pipeline until *all* visibility cubes are completed. Once all these visibility cubes are up to date, the visibility computation can be done.

The key point in these algorithms is obviously the $getAvailableGraphicsPipe()$ function that makes the process that calls it wait for a graphics pipeline to be free in order to make the desired operations on it. In addition to the two calls made in Algorithms 2 and 3, this function can also be called from inside a hybrid point-based visibility request which decided to enhance its pixmap resolution over a problematic place.

In an ideal world where each process would have its own graphics pipeline, this would not be problematic. However, in common configurations, the number of available graphics pipelines is lower than the number of processors. Thus, graphics pipelines become critical resources that must be shared between processes. In the worst cases, say for instance 1 graphics pipeline for 64 processors, processes will come to spend their time waiting for the critical resources instead of doing worthwhile computations, thus losing the benefits of hardware acceleration.

We have felt the need to enhance the previous algorithms in order to handle these problematic, but common, situations. The idea is very simple: instead of waiting for an available graphics pipeline until it is acquired, the $getAvailableGraphicsPipe()$ function makes a fixed number of attempts[3] to get one and returns an error code if it fails. In those cases, we may either retry to get a graphics pipeline or resort to a software-based solution: the $getVisibleSurfaces()$ function in Algorithm 2 or a BSP-based visibility request in the two other cases. A further enhancement could be to attribute the graphics pipelines in priority to Algorithm 2 in order to favor surface clipping operations.

6 Results

6.1 Experimental environment

Technical constraints

All the presented algorithms have been implemented inside the CANDELA platform, a research project designed to provide both a flexible architecture for testing and implementing new radiosity and radiance algorithms [26], and to handle real input data (complex geometrical surfaces, accurate light sources models, real spectrum modeling) in order to compute physically correct results.

The CANDELA software consists of approximatively 400 C++ classes and is based on the *Open Inventor* library. This enables us to obtain the required flexibility, both on inputs and algorithmic combinations. Parallelizing C++ algorithms inside such a large platform can seem challenging. It will be important not to deviate too far from the sequential code in order that we can still make valid comparisons and take advantage of the code's latest enhancements.

Hardware

The experiments were run on a SGI Origin2000 with 64 processors organized in 32 nodes. Each node consists of two $R10000$ processors with 32 K-Bytes of first level cache (L1) of data on the chip, 4 M-Bytes of external second level cache (L2) of data and instructions and 256 M-Bytes of local memory, for a total of 8 G-Bytes of physical memory. It was connected to a SGI InfiniteReality2 graphics pipeline with one raster manager.

Measures

The $R10000$ hardware performance counters combined with the software tool *perfex* allow performance measures of the behavior of the parallel program. This allows us to study, among others:

- *Speed-up*. This is defined by the ratio of the best sequential time over the parallel time obtained with n processors.
- *Memory overhead*. This is the ratio of time spent in memory over the total execution time.
- *L1 cache hit rate*. This is the fraction of data accesses which are satisfied from a cache line already resident in the primary data cache.
- *L2 cache hit rate*. This is the fraction of data accesses which are satisfied from a cache line already resident in the secondary data cache.

Moreover, we have instrumented the CANDELA libraries in order to monitor locks and synchronization times and to get information about the application behavior, in terms of handled surfaces or visibility requests.

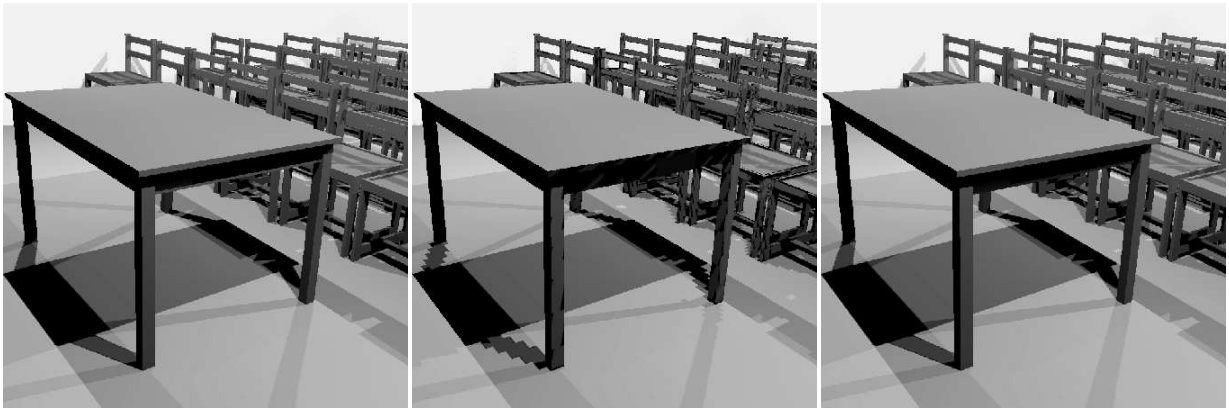
6.2 Performance evaluation

For the purpose of performance evaluation, we have performed several experiments on a single test scene: the classroom model provided by Peter Shirley. This scene consists of 3 153 initial surfaces and 4 light sources. We have chosen to consider:

- the hybrid hardware-based visibility algorithm,
  - with visible surfaces clipping either enabled or disabled,
  - with different pixmaps sizes: 40, 100, 200;
- the BSP-based visibility algorithm for comparison;
- 1 to 32 processors;
- a convergence rate[4] of about 90%;
- the inner loop parallel shooting algorithm: at each step the most energetic surface is shot by all processes, with a synchronization barrier at the end.

[3] In our current implementation, we use 10 attempts.

[4] Computed as $(1 - remaining\ energy)/initial\ energy$.



(a) BSP. (b) Hardware. (c) Hybrid.

Figure 5: Classroom images with our 3 visibility algorithms.

Results

Figure 5 presents final images of the radiosity simulation final results with our three different visibility algorithms. The solution shown in Figure 5-a has been computed with the BSP-based visibility algorithm. Aliasing effects clearly appear with the hardware visibility algorithm (Figure 5-b), despite the high pixmaps size of 500 being used. On the other hand, the hardware-based hybrid visibility algorithm, with a pixmaps size of only 100, yields a result very similar to the software-based one (Figure 5-c). The final result is composed of approximately 60 000 mesh elements.

The execution times for the different visibility parameters are shown on Figures 6-a and 6-b as a function of the number of processors. Figure 6-c shows the speed-up curves for the BSP-based algorithm and for the hybrid hardware-based one with a pixmaps size of 100 (the other curves for different pixmaps sizes are very similar).

The speed-up that we have achieved is not so impressive. In order to analyze this performance degradation, we have studied several indicators of the application behavior. As for the same algorithm using BSP-based visibility, the $L1$ and $L2$ cache hits (Figure 6-d) remain very high (between 97% and 99% for $L1$ and between 89% and 95% for $L2$). The total time lost on locks for accessing the graphics pipeline and other critical parts of the algorithms (such as the list of tasks or the *Open Inventor* data structures) can also not explain the degradation, although it is slowly increasing with the number of processors (Figure 6-e).

We also have tried to understand how the concurrent access to the graphics pipeline impacts on our visibility algorithm. For instance, Figure 6-f shows the ratio of visibility requests that could not be done by hardware for the different visibility parameters.

In order to understand these curves, we also give the total number of visibility requests that have been involved by each parameterization of the visibility algorithms (see Table 1). It is clear that the visible surfaces clipping step decreases the number of handled interactions between input surfaces, and thus the total number of visibility requests. This phenomenum is amplified as the pixmaps size decreases. Indeed, the use of smaller pixmaps sizes tends to lead to the missing of thin surfaces, thus decreasing both the number of handled interactions between input surfaces and the number of subdivisions when a small blocker falls between two surfaces.

Finally, it can be noted that the number of total visibility requests for the BSP-based visibility approach is very similar to that of the hardware-based hybrid visibility approach with a pixmaps size of 200. However, the difference is much greater for a pixmaps size of 100, even if the results are visually not different (see Figure 5).

Visibility	Handled surfaces	Requests
EV	157 600	27 758 156
AGL200 OFF	157 600	28 602 421
AGL200 ON	54 558	26 568 847
AGL100 ON	157 600	22 333 004
AGL100 OFF	47 100	19 994 829
AGL40 ON	157 600	16 132 843
AGL40 OFF	33 314	12 793 748

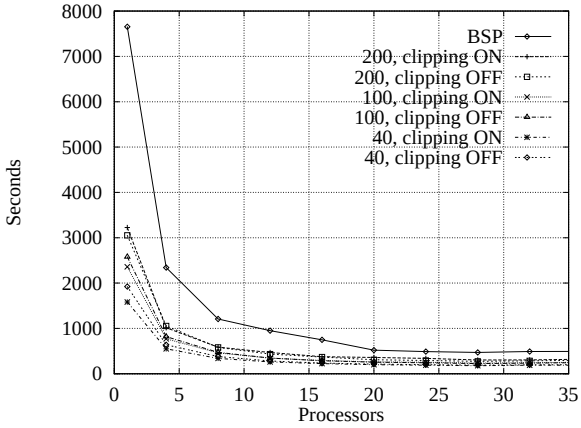
Table 1: Handled surfaces and visibility requests.

Discussion

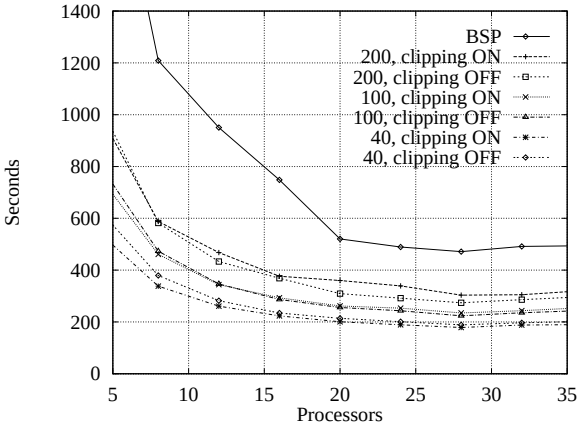
We can first note that using hardware acceleration greatly decreases the time needed to compute the radiosity solution, by a factor of two or three, for the same number of processors, up to 16. Using more processors, the hardware use is less interesting, but it is not so surprising because the achieved time becomes very low (less than 4 minutes for the whole simulation): it is not clear that the sequential part does not become dominant!

The ccNUMA architecture of the SGI Origin2000 appears to comfortably support our application. Indeed, the $L1$ and $L2$ cache hit rates remain very high and the ratio of time spent in memory over the total execution time decreases with the number of processors. Moreover, the visible surfaces clipping option of our hardware-based algorithm seems to enhance the natural data locality exhibited by the wavelet radiosity (Figure 6-d). Indeed, this allows us to quickly restrict the subset of surfaces to handle, that are likely to be located in a same small part of the memory.

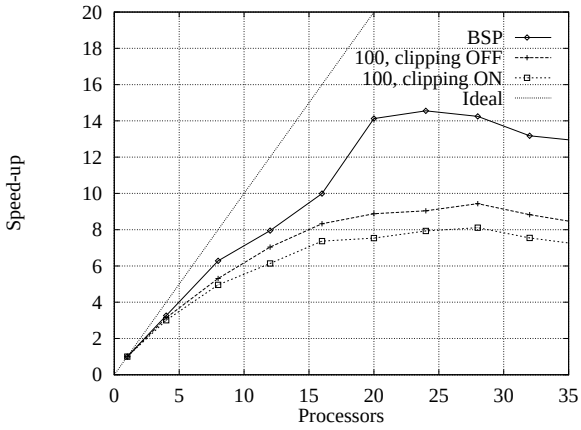
The interaction between graphics hardware acceleration and the behavior of our visibility algorithm executed in parallel is very sensitive to the parameters. Indeed, we would have suspected larger pixmaps sizes to be responsible for a larger ratio of hardware visibility requests to be rerouted to software-based visibility, due to their longer construction time. However, Figure 6-f shows the contrary! Actually, small pixmaps sizes cause a lower total number



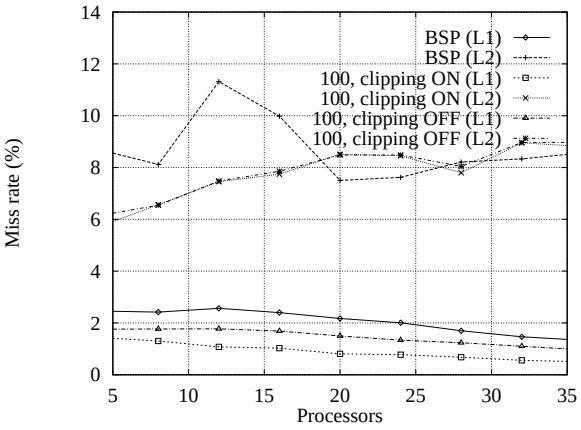
(a) Execution times.



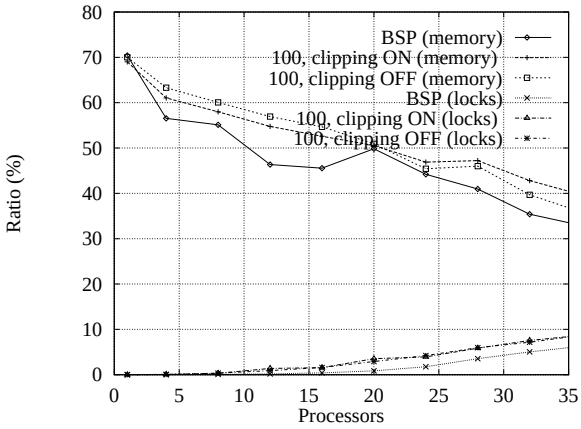
(b) Execution times: zoom in of (a).



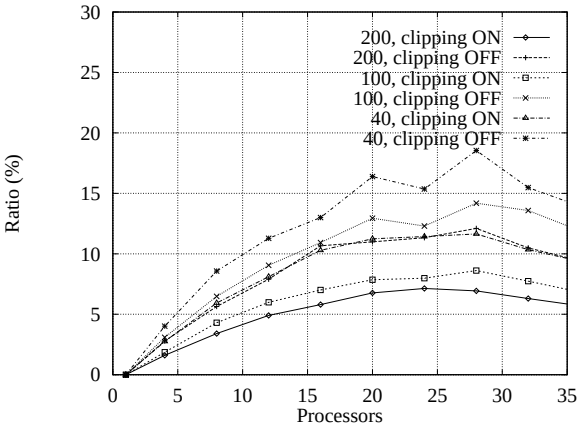
(c) Speed-up.



(d) Caches misses.



(e) Ratio of time spent in memory and locks.



(f) Ratio of hardware requests rerouted to software (due to busy graphics pipeline).

Figure 6: Classroom experimentation results

of visibility requests. Thus, the time needed to build the pixmaps becomes important compared to the visibility computations time, and a larger proportion of hardware visibility requests are blocked. Large pixmaps sizes lead to a large number of visibility requests, and thus to a better reuse of already build pixmaps.

The granularity we used for the parallelization is surely the most problematic aspect of this experimentation. Indeed, the inner loop parallelization has two drawbacks: first, the preparation (*i.e.* the push/pull of the energy at each level) of the surface with the most energy that will be handled in parallel has to be done sequentially. Second, a synchronization barrier is needed before the next surface to be handled. The time lost at these two points, as the number of processors increase, may be largely responsible for the performance degradation.

7 General discussion

Graphics hardware acceleration greatly increases the efficiency of our wavelet radiosity algorithm, and even more so when combined with multi-processing. Indeed, the high computational resources required by wavelet radiosity are mainly devoted to visibility requests which can be efficiently solved by hardware using dedicated graphics pipelines. Moreover, our hardware-based visibility algorithms greatly benefit from the spatial coherency of the involved visibility requests, thus allowing a large reuse of structures built with hardware.

An efficient hardware acceleration, however, can only be achieved if the application has direct access to the results of the computations made by the graphics pipelines. Recent advances in graphics architectures now allow such a direct connection[5]: on the SGI systems, this is done through a pixel buffer (*pbuffer* for short). If no direct connection is available, it is certainly better to turn toward a software-based solution!

Combined use of graphics hardware and multi-processing introduce a new bottleneck in algorithms relying on this scheme. Indeed, a given graphics pipeline cannot be accessed in parallel by several processes. During our tuning phase, we have noticed that waiting for the graphics pipeline to be free can be very time-consuming. In those cases, it is better, when possible, to resort to an equivalent software-based algorithm. When tasks of differing importance have to be performed by hardware, it could also be interesting to introduce a priority system. In our case, clipping visible surfaces must be done in priority before other visibility computations, because other processes may be waiting for the result of this computation. Of course, using a multi-pipelines configuration, these access conflicts will be reduced. In an ideal world, we could have one pipeline for one processor! Our experiments seem to show that one pipeline for 4 or 8 processors could be sufficient.

Finally, in order to fully benefit from the graphics hardware acceleration, the load-balancing of our parallel radiosity algorithm has to be improved in order to avoid unnecessary idle time (when waiting for a surface to be free) and synchronization barriers. Indeed, these phenomena are mainly responsible for our bad speed-up curves, which are still not really convincing.

Despite the early state of our work on this new promising combination, we were able to compute a radiosity solution of the Soda Hall building within 15 minutes for the direct illumination of the 98 point light sources and about 8 hours for both the 65 emitting surfaces and 100 shooting iterations, on 16 processors and a single graphics pipeline. Figure 7 and 8 show images of the obtained result.

[5] Actually, you may have to decrease the screen resolution in order to get a pbuffer visual for $3D$ computations.

8 Conclusion and future work

We have presented in this paper a new parallel wavelet algorithm that uses a novel combination of multi-processors and graphics hardware. The solution proposed consists of using the capabilities of graphics hardware for performing the costly geometric calculations, while the remaining computations are performed in parallel on the processors. Executing this program on the SGI Origin2000 with 32 processors and the SGI InfiniteReality2, we compute a highly precise radiosity solution in a near reasonable time. Moreover, our algorithm also benefits from the graphics pipeline for rendering interactive visualization at every step of the iterative resolution process. In a multi-processors/multi-pipelines environment, some processors could have a read-only access to the computation results in progress and have their own pipeline(s) for interactively displaying intermediate results. As a result, we think that our new scheme is a promising avenue to make 3D realistic rendering from geometric models tractable, at last, for real world applications. We also think that the combination that we propose could be used in some other applications. Physical simulation, such as heat transfer, acoustic simulation, N-body methods, *etc*, may also benefit from the combined multi-resolution approach and graphics hardware acceleration.

More performance evaluations are needed for tailoring our scheme to work well at different scales. Our goal in future experiments is to extend our approach from small scale (2 processors and 1 graphics pipeline) to large scale (p processors and n graphics pipelines) systems.

Acknowledgments

We would like to thank the Centre Charles Hermite for providing the resources necessary to the experimentations implied by our research work, and especially Alain Filbois for the time spent on the graphics system configuration, the members of the ISA team for their work on the CANDELA project and Carlo Sequin who provided the Soda Hall model.

References

[1] James Ahrens and James Painter. Efficient Sort-Last Rendering Using Compression-Based Image Compositing. In *Proceedings of the Second Eurographics Workshop on Parallel Graphics and Visualisation*, pages 145–154, Rennes, France, September 1998. Eurographics.

[2] Laurent Alonso and Nicolas Holzschuch. Using graphics hardware to speed-up your visibility queries. *Journal of Graphics Tools*, 1999. Submitted for publication. Also available as http://www.loria.fr/~holzschu/Publications/99-R-030.pdf.

[3] Daniel R. Baum and James M. Winget. Real Time Radiosity Through Parallel Processing and Hardware Acceleration. In *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, pages 67–75, March 1990.

[4] Xavier Cavin, Laurent Alonso, and Jean-Claude Paul. Parallel wavelet radiosity. In *Proceedings of the Second Eurographics Workshop on Parallel Graphics and Visualisation*, pages 61–75, Rennes, France, September 1998. Eurographics.

[5] Alan Chalmers and Erik Reinhard. *Parallel and Distributed Photo-Realistic Rendering*. ACM SIGGRAPH'98 Course Notes, July 1998. Course 3.

[6] Michael Cohen, Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A Progressive Refinement Approach to Fast Radiosity Image Generation. In *Computer Graphics (ACM SIGGRAPH '88 Proceedings)*, volume 22, pages 75–84, August 1988.

[7] Michael Cohen and Donald P. Greenberg. The Hemi-Cube: A Radiosity Solution for Complex Environments. In *Computer Graphics (ACM SIGGRAPH '85 Proceedings)*, volume 19, pages 31–40, August 1985.

[8] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, Boston, MA, 1993.

[9] Commission Internationale de l'Éclairage, Bureau de la CIE, Paris. *Recommended File Format For Electronic Transfer Of Luminaire Photometric Data*, 1993. CIE 102.

[10] Marc de Berg. Linear Size Binary Space Partitions for Fat Objects. In *Proceedings of the 3rd Annual European Symposium on Algorithms*, pages 252–263, 1995.

[11] Thomas A. Funkhouser. Coarse-Grained Parallelism for Hierarchical Radiosity Using Group Iterative Methods. In *Computer Graphics Proceedings, Annual Conference Series, 1996 (ACM SIGGRAPH '96 Proceedings)*, pages 343–352, 1996.

[12] Steven J. Gortler, Peter Schroder, Michael F. Cohen, and Pat Hanrahan. Wavelet Radiosity. In *Computer Graphics Proceedings, Annual Conference Series, 1993 (ACM SIGGRAPH '93 Proceedings)*, pages 221–230, 1993.

[13] Dongming Jiang and Jaswinder Pal Singh. A Methodology and an Evaluation of the SGI Origin2000. In *ACM Sigmectrics/Peformance*, Madison, Wisconsin, June 1998.

[14] Alexander Keller. Instant radiosity. In *Computer Graphics (ACM SIGGRAPH '97 Proceedings)*, volume 31, pages 49–56, 1997.

[15] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, Denver, June 1997. ACM Press.

[16] I. Martin, X. Pueyo, and D. Tost. A two-pass hardware-based method for hierarchical radiosity. *Computer Graphics Journal (Proc. Eurographics '98)*, 17(3):C159–C164, September 1998.

[17] D. Meneveaux, K. Bouatouch, and E. Maisel. Memory management schemes for radiosity computation in complex environments. In *Proc. Computer Graphics International '98 (CGI '98)*, Hannover, Germany, June 1998.

[18] Steven Parker, William Martin, Peter-Pike Sloan, Peter Shirley, Brian Smits, and Chuck Hansen. Interactive Ray Tracing. In *I3D*, April 1999. Accepted for publication.

[19] Luc Renambot, Bruno Arnaldi, Thierry Priol, and Xavier Pueyo. Towards efficient parallel radiosity for dsm-based parallel computers using virtual interfaces. In *Proceedings of the Third Parallel Rendering Symposium (PRS '97)*, pages 79–86, Phoenix, AZ, October 1997. IEEE Computer Society.

[20] Luc Renambot and David Figuls. Convergence analysis in a parallel radiosity algorithm using virtual interface. In *Proceedings of the Second Eurographics Workshop on Parallel Graphics and Visualisation*, pages 31–48, Rennes, France, September 1998. Eurographics.

[21] Peter Schroder. *Wavelet Algorithms for Illumination Computations*. Ph.D. thesis, Technical Report, Princeton, NJ, November 1994.

[22] Peter Schroder, Steven J. Gortler, Michael F. Cohen, and Pat Hanrahan. Wavelet Projections for Radiosity. In *Fourth Eurographics Workshop on Rendering*, number Series EG 93 RW, pages 105–114, Paris, France, June 1993.

[23] Peter Schroder, Steven J. Gortler, Michael F. Cohen, and Pat Hanrahan. Wavelet Projections for Radiosity. *Computer Graphics Forum*, 13(2):141–151, June 1994.

[24] Francois Sillion and Claude Puech. *Radiosity and Global Illumination*. Morgan Kaufmann, San Francisco, CA, 1994.

[25] Jaswinder Pal Singh, John L. Hennessy, and Annoop Gupta. Implications of Hierarchical N-body Methods for Multiprocessor Architectures. *ACM Transactions on Computer Systems*, 13(2):141–202, May 1995.

[26] Christophe Winkler. *Expérimentation d'algorithmes de calcul de radiosité à base d'ondelettes*. PhD thesis, Institut National Polytechnique de Lorraine, 1998.

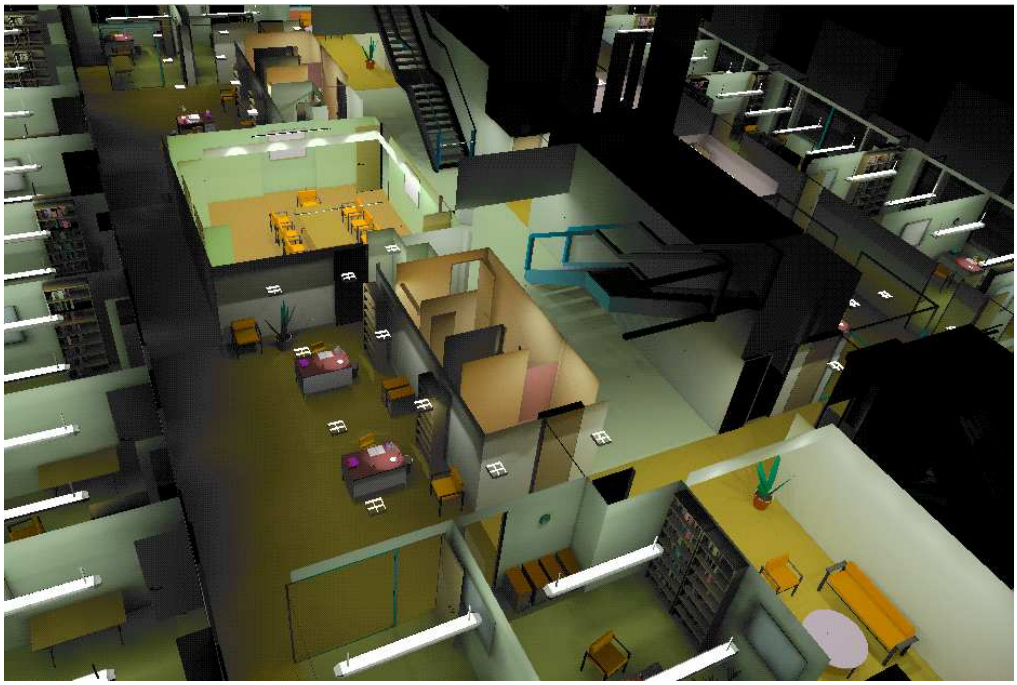
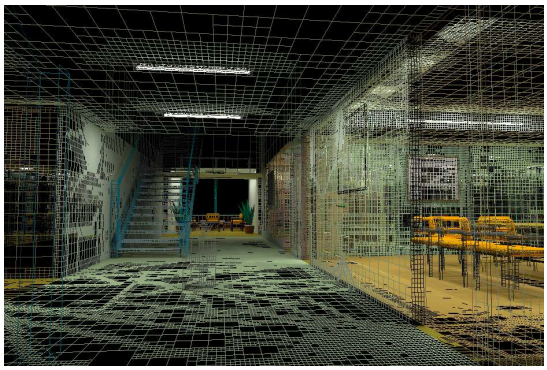


Figure 7: Soda Hall final image.

(a) Final image (no texture).



(b) Finite element decomposition.

(c) Final image (with texture).

Figure 8: Soda Hall corridor view.